NASA Technical Memorandum 106180

A Multiarchitecture Parallel-Processing Development Environment

Scott Townsend
Sverdrup Technology, Inc.
Lewis Research Center Group
Brook Park, Ohio

and

Richard Blech and Gary Cole
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio

Prepared for the
Seventh International Parallel Processing Symposium
sponsored by the Institute of Electrical and Electronics
Engineers Computer Society
Newport Beach, California, April 13–16, 1993

NASA

A Multiarchitecture Parallel-Processing Development Environment

Scott Townsend *
Sverdrup Technology, Inc
Lewis Research Center Group
Brook Park, Ohio 44142

Richard Blech and Gary Cole
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

Abstract

A description is given of the hardware and software of a multiprocessor test bed — the second generation Hypercluster system. The Hypercluster architecture consists of a standard hypercube distributed-memory topology, with multiprocessor shared-memory nodes. By using standard, off-the-shelf hardware, the system can be upgraded to use rapidly improving computer technology. The Hypercluster's multiarchitecture nature makes it suitable for researching parallel algorithms in computational field simulation applications (e.g., computational fluid dynamics). The dedicated test-bed environment of the Hypercluster and its custom-built software allows experiments with various parallel-processing concepts such as message passing algorithms, debugging tools, and computational "steering". Such research would be difficult, if not impossible, to achieve on shared, commercial systems.

Keywords — parallel processing, system software, computer architecture, computational fluid mechanics.

1 Introduction

For over ten years, the NASA Lewis Research Center has been developing test-bed systems for researching the hardware and software aspects of parallel processing. Early efforts [1–6] focused on using multiple microprocessors to achieve low-cost, real-time simulation of airbreathing propulsion systems described by systems of ordinary differential equations. Since then, attention has turned to computational field simulation, involving the solution of systems of partial differential equations (e.g., computational fluid dynamics (CFD)).

Because field simulation can involve multiple levels of parallelism, the Hypercluster was conceived as a multiarchitecture test bed [7] to research parallel processing issues concerning CFD algorithms and applications, as well as system software tools and utilities. The system consists of a front-end processor connected to a network of nodes which are arranged in a hypercube distributed-memory topology. Each node can have multiple processors connected through shared memory, and multiple communication links to other nodes in the system. Applications are written with explicit message passing and/or shared memory constructs. The parallel processing library developed to support these constructs on the Hypercluster is described in [8]. A message passing kernel and an initial operating capability for the original Hypercluster system are described in [9] and [10], respectively.

This report describes the current, second generation Hypercluster. The system hardware is described first, followed by a discussion of the system software and software development of applications. Finally, a brief overview of application and system software research conducted on the Hypercluster is given.

*This work was supported by the NASA Lewis Research Center under contract NAS3-25266 with Gary Cole as monitor.

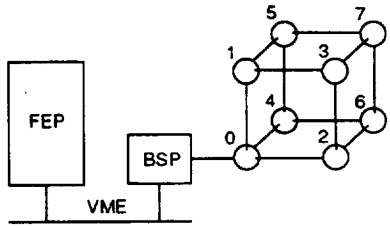

Figure 1: System topology

2 System hardware

The system is built exclusively from off-the-shelf hardware, described below.

2.1 Front-end processor (FEP)

At the highest level, the hardware consists of a Front-End Processor (FEP) connected to a network of multiprocessor nodes (see Figure 1).

The FEP is a Silicon Graphics Personal Iris which runs a version of UNIX. It communicates with the network of nodes via an intermediary Backplane Service Processor (BSP). Communication between the FEP and BSP occurs over a VME to VME bus repeater. Communication between the BSP and the network of nodes occurs over a dual-port memory communication link identical to the links (described below) used between nodes.

2.2 Network processors

There are a total of 40 processors in the network, of which 32 are used for application processing. The

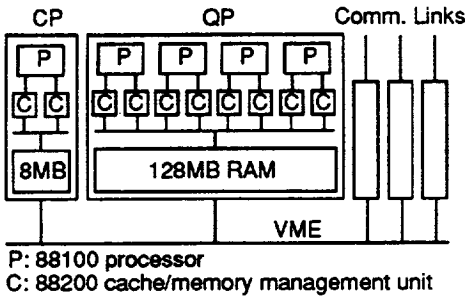

Figure 2: Node architecture

network consists of eight nodes arranged in a hypercube topology. Each node contains a Communications Processor (CP), a Quad Processor (QP), and three communication links (see Figure 2). Node zero has an extra communication link for communicating with the BSP. Physically, each node is in a separate VME card cage. The CP and QP within a node communicate via shared memory on the VME bus.

The CP handles all communications between nodes independently of the applications running on the QP. It can also be used for application processing, though this will detract from overall performance if the application is communication-bound. The CP is a Motorola 88000 processor (described below) with 8MB of memory. The QP is used for application processing only. It consists of four 88000 processors sharing a total of 128MB of cache-coherent memory. This memory is normally accessed from a private bus, so there is little interaction with VME traffic resulting from node-to-node communications.

Each Motorola 88000 processor used in the CP and QP modules is a RISC (Reduced Instruction Set Computer) consisting of one 88100 microprocessor and two 88200 cache/memory management chips. The 88100 contains 32 32-bit registers, five independent execution units (instruction fetch, data load/store, integer operations, floating-point add operations, and floating-point multiply operations), and is heavily pipelined. Each 88200 chip contains 16KB of cache and the logic required for demand-paged virtual memory. There are two 88200s per 88100 to support simultaneous instruction and data transactions.

2.3 Communication links

The communication links between nodes are Bit3 Corporation VME to VME adaptors with Direct Memory Access (DMA) capability. Each pair of boards (one in each node's card cage) have a cable and 128KB of dual-port memory between them. Messages are passed between nodes as the source CP copies a packet to the dual-port memory and then sends an interrupt to the destination node. The destination CP will then copy the message out of the dual-port memory and process it. Due to hardware limitations in the links, the DMA circuitry can only be used for one of the packet copy operations.

The link dual-port memory is evenly split between the two connected nodes, which use their assigned half for outbound packets and receive packets from the other node's half. VME bus contention is reduced on the receiving node by using the dual-port memory rather than directly manipulating the receiver's VME memory.

The link dual-port memory could also be used as a (small) memory shared between application processes. Future research may look into what advantages might result from such usage.

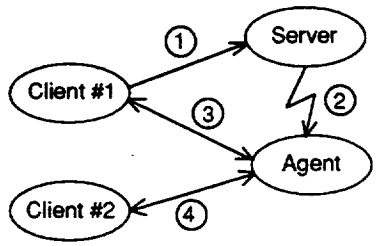

Figure 3: Client — server processes

3 System software

The system software is split between front-end software which runs on the FEP and user workstations, and a Message Passing Kernel (MPK) which runs on the CP and QP processors. The front-end software is written using a client-server model. The server code must run on the FEP, while the client code can be run from the FEP or any workstation on the Internet. Different client programs support different styles of user interaction, all accessing the same server.

3.1 Server software

The server software is responsible for providing local or remote client access, local client file I/O, system configuration, kernel and application loading, and system health checks. (Local clients are client processes which run on the FEP processor, remote clients access the FEP via the Internet). The server also acts as a gateway between the client and the application code running on the network processors. The server accepts requests from either a UNIX domain socket for local clients or an Internet domain socket for remote clients. Client login follows a three step process as shown in Figure 3:

1. The client sends a login request to the well-known server socket.
2. If the system is in use, the server sends a reply to the client identifying the current user and rejects the login. If the system is available, the server will *fork* a copy of itself, known as the *agent* process. This agent process will then reply to the client with a message indicating that the login has been accepted. Once the agent is started, the server resumes monitoring its well-known port for login requests from other users.
3. The client proceeds to issue requests and receive replies with the agent process.

If the client requires multiple processes, the additional processes can register with the agent process (shown as step 4 in Figure 3). The registration scheme allows additional client processes to be used as message handlers.

For local clients, the agent process will handle all file I/O requests made by the application directly, except for I/O to the user's terminal. For remote clients or local client terminal I/O, the agent will forward the I/O request on to the client. In this way all application I/O occurs in the user's local file system even if the user is running a client from a remote workstation. Applications which have heavy I/O traffic will achieve higher performance when run from a local client. With up to 32 application processors running, it is quite possible to exceed the agent's (or remote client's) maximum number of open files. To avoid this problem, the agent (and remote client) code employs a file descriptor caching scheme to provide an unlimited number of open files.

The agent allows the user to configure the kernels to be loaded on the BSP, CPs, and QPs and perform a system reboot. In this way new kernels can be debugged by ordinary clients without locking-out other users between runs. (At each user logout the system is rebooted with the standard kernels in a procedure that takes under 20 seconds.) The user may also configure how many processors in the QP module should actually be used. By changing the number of active QP module processors, the user can alter how much memory an application may have per processor from 32MB to 128MB. Thus the system can be arranged in configurations which vary between 32 processors of 32MB each to 8 processors of 128MB each. This capability has been found useful in the initial porting stages of large sequential codes.

The agent periodically checks the health of the system by sending a short message to each processor. Upon reception, the destination processor simply echoes the message back to the agent. If the agent does not see the echo within a timeout period, the failed processor is noted and the system automatically reboots. These health check messages are limited to one per second to minimize disruption of a running application. During kernel debugging a typical failure mode is for messaging between nodes to become deadlocked, and these health checks will detect this condition. While running applications it is rare for the system to fail in this manner, though it is still possible with a "runaway" application since the kernel does not enforce flow control.

Whenever the agent receives a message from the system which is not intended for it, it will forward this message to the client via the UNIX domain socket or the Internet domain socket depending upon the type of client. Multiple process clients will have the message forwarded to the appropriate client process depending on the type of message and which process registered as a handler for that type. If the agent receives a message from the client which is destined for a processor in the network of nodes, it is forwarded via the BSP to the destination. This gateway function allows arbitrary communication between (possibly remote) clients and the application running in the system.

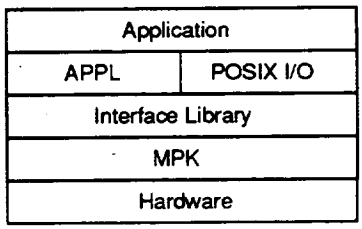

Figure 4: Software layers

3.2 Client software

To support writing local or remote clients, a set of high-level routines is provided in a library. This library provides support for accessing the server, sending commands, receiving replies, handling file I/O messages from the application, searching the application symbol table, manipulating application variables, and communicating with application processors. In addition, a symbolic traceback is provided automatically upon any abnormal application exit. Using this library, simple clients such as a line-oriented debugger take only a few pages of C code.

Existing clients include the above line-oriented debugger, a menu-oriented debugger, a client for running applications written with the Application Portable Parallel Library (APPL, [11], discussed later), a client which gathers system statistics and displays the results in various charts, and a few application-specific clients (see the "Parallel-Processing Software Research" section).

3.3 Message passing kernel (MPK)

The MPK supports a single task with primitives for explicit message passing, POSIX-style I/O, and shared memory operations within a node. These facilities are accessed through a library of interface routines. Normally the application will use APPL as a portable layer between it and the Hypercluster-specific routines of the MPK. Figure 4 depicts the layering between the application and the MPK. The current implementation is very similar in concept to that of the previous version [9], with changes made to support the new hardware. The previous version was written in assembler code to optimize speed, while the current version is written primarily in 'C', with only hardware interface and buffer copy routines written in assembly. A limited number of experiments have shown the C compiler to be sufficiently close to the efficiency of an assembler version to justify the convenience of a higher level language.

The shared memory operations supported by the MPK include barriers and semaphore operations. Routines are also available for determining the system configuration, the applications' node and processor number, and the system time.

Message passing primitives include buffered and non-buffered send, broadcast, blocking and non-blocking receive, and direct remote memory manip-

ulation (used for loading and debugging). Messages are routed based upon destination node and processor number. Messages within a node are sent taking advantage of the node's shared memory, while those destined outside the node will traverse one or more communication links in a simple store-and-forward fashion. This process could potentially include the Internet for a message intended for a remote client process. Large messages (larger than the current maximum packet size of 4KB) are split into packets before being sent outside the node. Due to deterministic routing of packets, message reassembly at the receiver is trivial. (Forwarding across the Internet is done via TCP, so remote client processes will still see a serial stream of MPK packets).

Application messages may have a type assigned to them by the user. Messages of the same type are held in separate queues by the receiving processor's MPK until the receiving application performs a receive operation requesting the corresponding type. A special value may be used to receive the next message of any type.

There is no flow control enforced by the MPK messaging scheme. For the applications studied so far this has not been a problem. Computations typically proceed through an iteration, exchange data, and begin the next iteration cycle. This style of algorithm tends to be self-pacing. Adding flow control to the MPK in the future is under consideration.

4 Application software

Hypercluster applications are written in standard sequential FORTRAN and/or C and are compiled using a commercial cross-compiler running on a Sun SPARCstation. Full runtime libraries for both languages are provided, making the porting process of existing sequential codes to a single Hypercluster processor straightforward.

All parallel constructs must be explicitly controlled by the programmer; the compilers do not support any parallel extensions or perform any automatic parallel optimizations. An interface library is used to access the facilities provided by the MPK. In addition, a port of APPL is available and is the preferred mechanism for using the parallel processing capabilities of the system. Codes using APPL are portable to a wide range of machines besides the Hypercluster.

5 Parallel-processing software research

The Hypercluster has been used to support a number of software research activities. A few examples of these are briefly described in the following sections.

5.1 Applications

Due to the difficulty of obtaining reliable access to a commercial parallel processor system at Lewis, the Hypercluster was used to develop and debug a parallel version of a large, 3-D CFD turbomachinery code referred to as ISTAGE [12]. The parallel version was programmed using the APPL, and was subsequently ported to commercial MIMD machines without modification. Figure 5 displays the speedup characteristics of this application on the Hypercluster and Intel iPSC/860. A viscous version of the code, MSTAGE,

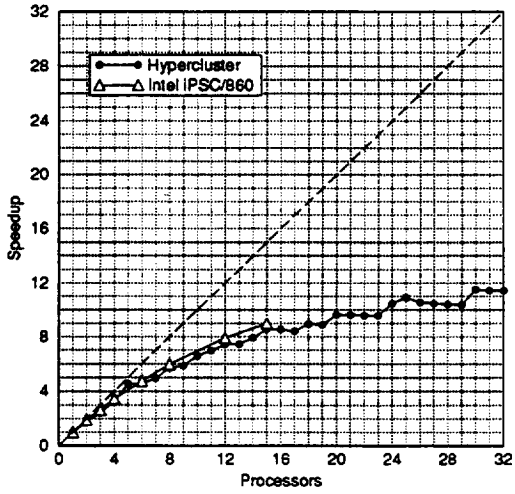

Figure 5: ISTAGE speedup

is now being parallelized for the Hypercluster in anticipation of its use with the Integrated CFD and Experiment (ICE) Project [13].

A trivial fractal application was written to demonstrate the interactive capabilities of Hypercluster client programs. The user specifies on the screen what area of the Mandelbrot set is to be evaluated and the client sends messages to Hypercluster processors to perform the evaluation. The client performs load balancing by giving each processor one row of the display to calculate; when a processor replies with its results, it is given the parameters of the next unevaluated row. This "embarassingly" parallel application also demonstrates the peak Hypercluster computational capability of 240 MFLOPS when using all 40 processors.

For a more representative view of the Hypercluster's performance, the nCUBE version of the SLALOM benchmark was ported and run [14]. Only 16 of the available 32 processors were used since this parallel version of SLALOM assumes a square hypercube topology. The system was able to evaluate 933 patches in the required 60 seconds, corresponding to a rate of 14.7 MFLOPS. This level of performance was achieved without any special coding tricks or optimizations. Tuning to the 88100 architecture should improve this figure.

5.2 System software

The Hypercluster played a significant role in the development and debugging of the APPL. The initial target machines for the APPL project were the Intel iPSC/860, the Alliant FX/80, and the Hypercluster. These represent three different classes of architectures: a distributed memory machine, a shared memory machine, and a machine which has a combination of shared and distributed memory. Although the Hypercluster and the Intel iPSC/860 were similar, the differences in the two architectures required developing a programming model which allowed an application to run on either machine without modification. This resulted in the APPL process definition

concept, which makes an APPL program "architecture independent".

In the area of performance visualization, a special client sub-process has been written to gather and display statistics for the system while it is running an application. Statistics such as CPU usage, message communication rates, and link communication rates for the entire system are shown as bar graphs in the main panel. A more detailed view of a given processor or link can be called up by clicking on the corresponding chart in the main panel with the mouse. The detailed displays show trends over the last thirty seconds along with numerical totals. Figure 6 shows the display during a run of the SLALOM benchmark described above. Performance problems such as load imbalances or link congestion can be seen with a glance at the main panel. This capability is available to any client running on a Silicon Graphics workstation by linking with the standard client support library.

As a first step towards system support capable of "steering" a computation, a sample application-specific client has been written to interactively view and manipulate a calculation as it progresses (see Figure 7). This client uses the user's workstation for graphical visualization of results as they are computed. Results may also be recorded for later playback. Coefficients used in the algorithm may be altered in real-time and the effect on the solution process is displayed. Currently only the coefficients are directly manipulated by the client; the application running in the Hypercluster processors is responsible for distributing the initial temperatures and assembling the (distributed) solution data into BSP memory, which the client then accesses for display. With additional work in the area of describing how application data is distributed among processors, it would be possible for the client to distribute the initial temperature data and assemble the solution data directly from the Hypercluster processors for display. This approach may be investigated as part of future system software research activities.

6 Concluding remarks

The above description of the second generation Hypercluster shows it to be a parallel processor based upon hypercube topology, with each node a shared memory multiprocessor in its own right. The new hardware uses contemporary RISC processors with a front end computer running UNIX.

The dedicated test-bed environment and custom-built software make the Hypercluster ideal for parallel-processing research in both the application and system software arenas. The Hypercluster has been used to conduct software experiments that would have been difficult or even impossible to do on a shared, commercial system.

Finally, the system continues to evolve. Possible future work includes investigations into better support for application debugging, performance visualization, and the interactive steering of a computation in progress.

References

[1] Blech, R. A. and Arpasi, D. J.: "Hardware for a Real-Time Multiprocessor Simulator." NASA TM 83805, 1985.

[2] Blech, R. A. and Williams, A. D.: "Hardware Configuration for a Real-Time Multiprocessor Simulator." NASA TM 88802, 1986.

[3] Arpasi, D. J.: "RTMPL–A Structured Programming and Documentation Utility for Real-Time Multiprocessor Simulations." NASA TM 83606, 1984.

[4] Cole, G. L.: "Operating System for a Real-Time Multiprocessor Propulsion System Simulator." NASA TM 83605, 1984.

[5] Arpasi, D. J. and Milner, E. J.: "Partitioning and Packing Mathematical Simulation Models for Calculation on Parallel Computers." NASA TM 87170, 1986.

[6] Milner, E. J. and Arpasi, D. J.: "Simulating a Small Turboshaft Engine in a Real-Time Multiprocessor Simulator (RTMPS) Environment." NASA TM 87216, 1986.

[7] Blech, R. A.: "The Hypercluster: A Parallel Processing Test-Bed Architecture for Computational Mechanics Applications." NASA TM-89823, 1987.

[8] Quealy, A.: "Hypercluster Parallel Processing Library User's Manual." NASA CR 185231, 1990.

[9] Blech, R. A.; Quealy, A.; and Cole, G. L.: "A Message-Passing Kernel for the Hypercluster Parallel-Processing Test Bed." NASA TM-101952, 1989.

[10] Cole, G. L.: Blech, R. A.; and Quealy, A.: "Initial Operating Capability for the Hypercluster Parallel-Processing Test Bed." NASA TM-101953, 1989.

[11] Quealy, A.: "Portable Programming on Parallel/Networked Computers Using the Application Portable Parallel Library (APPL)." to be published.

[12] Blech, R.A.; Milner, E. J.; Quealy, A.; and Townsend, S.E.: "Turbomachinery CFD on Parallel Computers." NASA TM-105932

[13] Szuch, J. R.; and Arpasi, D. J.: "Enhancing Aeropropulsion Research with High-Speed Interactive Computing." NASA TM 104374, 1991.

[14] Gustafson, J., et al.: "The Design of a Scalable, Fixed-Time Computer Benchmark." *J. of Parallel and Distributed Computing 12*,388-401, 1991.

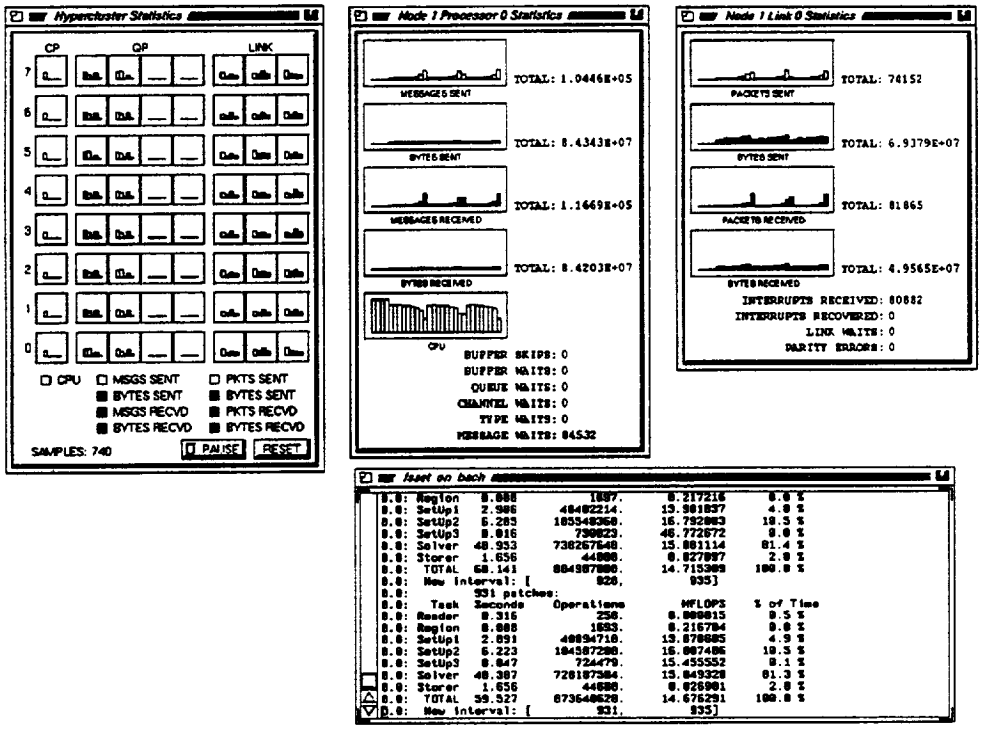

Figure 6: Runtime statistics during SLALOM benchmark

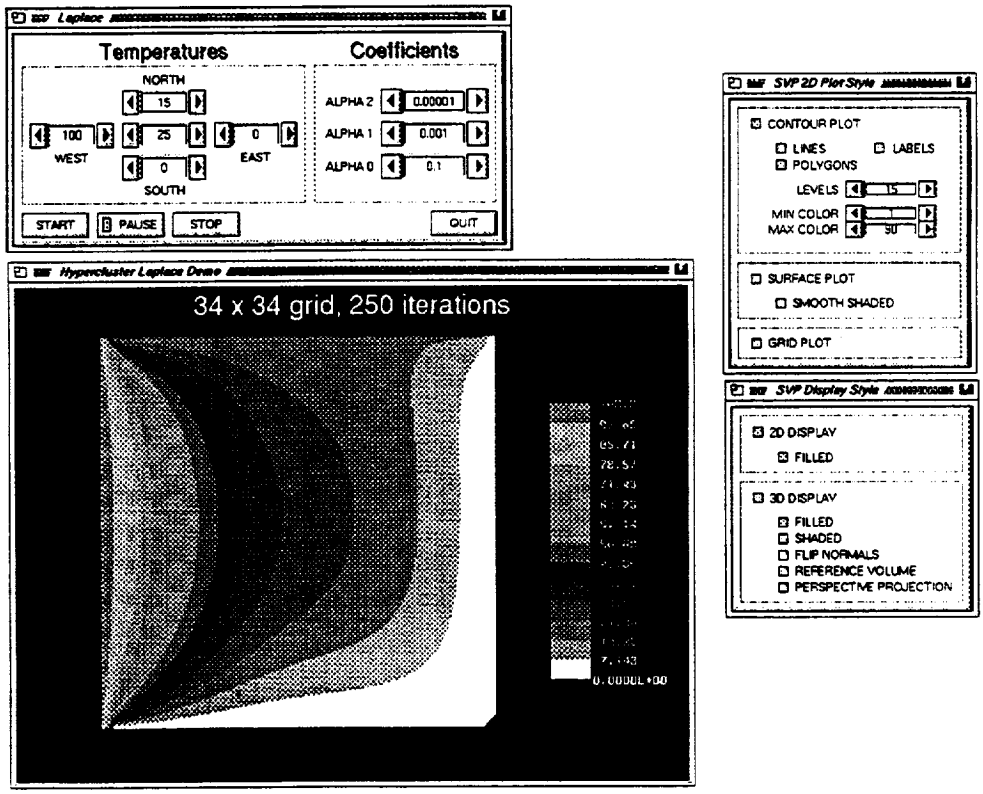

Figure 7: Visualization and steering of a computation

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1993	Technical Memorandum

4. TITLE AND SUBTITLE

A Multiarchitecture Parallel-Processing Development Environment

5. FUNDING NUMBERS

WU-505-62-52

6. AUTHOR(S)

Scott Townsend, Richard Blech, and Gary Cole

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E-7744

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, D.C. 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM-106180

11. SUPPLEMENTARY NOTES

Prepared for the Seventh International Parallel Processing Symposium sponsored by the Institute of Electrical and Electronics Engineers Computer Society, Newport Beach, California, April 13-16, 1993. Scott Townsend, Sverdrup Technology, Inc., Lewis Research Center Group, 2001 Aerospace Parkway, Brook Park, Ohio 44142 and Richard Blech and Gary Cole, NASA Lewis Research Center, Cleveland, Ohio. Responsible person, Scott Townsend, (216) 433-8101.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT (***Maximum 200 words***)

A description is given of the hardware and software of a multiprocessor test bed - the second generation Hypercluster system. The Hypercluster architecture consists of a standard hypercube distributed-memory topology, with multiprocessor shared-memory nodes. By using standard, off-the-shelf hardware, the system can be upgraded to use rapidly improving computer technology. The Hypercluster's multiarchitecture nature makes it suitable for researching parallel algorithms in computational field simulation applications (e.g., computational fluid dynamics). The dedicated test-bed environment of the Hypercluster and its custom-built software allows experiments with various parallel-processing concepts such as message passing algorithms, debugging tools, and computational "steering". Such research would be difficult, if not impossible, to achieve on shared, commercial systems. Keywords - parallel processing, system software, computer architecture, computational fluid mechanics.

14. SUBJECT TERMS

Parallel processing; System software; Computer architecture; Computational fluid mechanics

15. NUMBER OF PAGES

7

16. PRICE CODE

A02

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	